Employing OpenCL as a Standard Hardware Abstraction in a Distributed Embedded System: A Case Study

Omair Rafique and Klaus Schneider
Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany
{rafique,schneider}@cs.uni-kl.de

***Abstract*—The open computing language (OpenCL) is a standard open source specification for parallel computing on heterogeneous architectures. OpenCL offers a set of abstract models for substantial acceleration in parallel computing and is supported by most of the leading hardware vendors. In this paper, we present a systematic approach for employing OpenCL as a hardware abstraction layer that enables the user to utilize the supported computing resources by using different scheduling and mapping schemes. We illustrate the approach by a case study of a distributed automotive embedded system. In particular, we developed an extendable set of related advanced driving assistance system (ADAS) applications under a common application setup which is mapped and executed on different OpenCL supported devices of an embedded platform. The detailed evaluations performed using different scheduling schemes in conjunction with various OpenCL mapping configurations are presented.**

***Index Terms*—Distributed embedded system, parallel computing, performance evaluation**

I. INTRODUCTION

A distributed embedded system (DES) is typically comprised of multiple control units (nodes) which are connected by a communication medium (network) to cooperate with each other to achieve a common goal [1]. Different control units perform different dedicated operations where each one as a subsystem performs some part of the overall system. The control unit reads out data from the sensor(s), performs required computations and generates the corresponding control signals/outputs to actuator(s). The control unit can feature a custom processor or a micro-controller or even a heterogeneous computing platform consisting of different devices like multicore processors, graphical processing units (GPUs), and other processor architectures.

A considerable effort has been made in introducing programming languages for the abstraction and better resource utilization of heterogeneous architectures. Among them, the open computing language (OpenCL) has found a lot of interest in heterogeneous computing and is supported by most of the leading hardware vendors including Intel, Apple, AMD, ARM and many others. In contrast to proprietary specification languages with limited hardware choices, OpenCL offers task-parallel and data parallel heterogeneous computing on a variety of modern CPUs, GPUs, digital signal processors (DSPs), and other microprocessor architectures [2].

OpenCL has already been thoroughly evaluated for general purpose computing with a variety of benchmarks. Despite the fact that OpenCL has shown promising results for substantial acceleration in parallel computing as presented in [3]–[5], it has not yet been systematically employed and evaluated for embedded or distributed systems. This is mainly because of a lack of support for OpenCL on devices typically used in embedded computing.

In this paper, we present a systematic approach that employs OpenCL as a standard hardware abstraction to allow the embedded system developer to explore and to evaluate the utilization of parallel computing resources using different scheduling and mapping schemes. Hence, enabling the user to potentially exploit both coarse-grained task-level parallelism and fine-grained data-level parallelism, respectively. In contrast to existing approaches that are limited to general purpose computing, we consider a case study of a distributed automotive embedded system, namely the *ConceptCar* [6], [7]. Specifically, the proposed approach is validated by evaluating OpenCL for embedded computing on a particular control unit of the ConceptCar. The computations are provided by an extendable set of related advanced driving assistance system (ADAS) [8] applications built under a common application setup. This application set is mapped and executed using different scheduling schemes in conjunction with various OpenCL mapping configurations.

II. PRELIMINARIES

This section first discusses OpenCL in general, and then presents the existing related approaches used for evaluating OpenCL for parallel computing.

A. Open Computing Language (OpenCL)

OpenCL [2] is an open specification language designed for parallel computing on cross-vendor and heterogeneous architectures. A primary benefit of OpenCL is a substantial acceleration in parallel processing. To this end, it supports both coarse-grained (task-level) as well as fine-grained (data-level) parallelism. Second, it provides the ability to write vendor-neutral cross platform applications. This is achieved by providing high-level abstractions to hide low-level details of implementations, such as drivers and runtime. The basic strength of this abstraction is the ability to scale code from simple embedded microcontrollers to multicore CPUs and highly parallel GPUs

Authorized licensed use limited to: University of Glasgow. Downloaded on August 30,2023 at 12:46:10 UTC from IEEE Xplore. Restrictions apply.

without revising the code. These benefits can be derived by understanding and exploiting a set of abstract models provided by OpenCL, as depicted in Fig. 1.

1) Platform Model: The OpenCL platform model provides users with a convenient abstraction of the target hardware. It is defined as a *host* connected to one or more compute devices, each having multiple compute units (CUs), each of which further consists of multiple processing elements (PEs).

A host is typically a CPU running a standard operating system (OS), while a compute device may be a GPU, a DSP, a further multicore CPU or any other specific microprocessor. Each device therefore consists of a collection of one or more CUs where each CU can be conceived as, for instance, a core of a CPU, or a streaming multiprocessor of a GPU. A CU is further composed of one or more PEs that execute instructions. Each PE can therefore be conceived as, for instance, a streaming core/SIMD lane of a GPU.

2) Program Model: The OpenCL program model is comprised of two main components: the *host program* and *kernels*. The host program resides and executes on the host and is responsible for setting up and handling the execution of kernels on the compute devices through command queues. Each command queue can represent a complete device (e.g., a CPU) or even a compute unit of that device (e.g., a CPU-core). Each command queue accepts the execution and memory requests for the corresponding devices and their CUs.

A kernel is a C-like function that actually implements the abstract behavior of the system or part of the system. The host program determines the fine-grained data-level parallelism by specifying the total number of execution instances when the kernel is placed on the command queue. Each independent instance is called a work-item that executes the same kernel function but on different data. These work-items that represent the execution instances of the associated kernel can be further grouped together in different work-groups as shown in Fig. 1.

3) Execution Model: The OpenCL execution model can be simply understood as the mapping of kernels on the platform model which is implemented in the host program. Depending on the target compute device (e.g., a CPU or a GPU), kernels are mapped differently. In case of GPUs, OpenCL only allows the user to create a command queue at the level of a compute device. Hence, for a GPU, a kernel is typically allocated on a compute device, a work-group is ideally mapped on a CU, and work-items of that work-group are executed by PEs of that CU, as depicted in Fig. 1. In contrast, for CPUs, a command queue can be created at the level of a compute device as well as at the level of a CU. For the latter, the whole kernel (all work-groups) are mapped to the same CU (i.e., a core of a CPU). Furthermore, the execution model also facilitates the usage of different scheduling schemes by allowing in-order and out-of-order execution of kernels.

B. Related Approaches

The main content of this paper is the evaluation of OpenCL for embedded computing by means of a case study. We therefore list related work on the general evaluation of OpenCL as presented in [3]–[5].

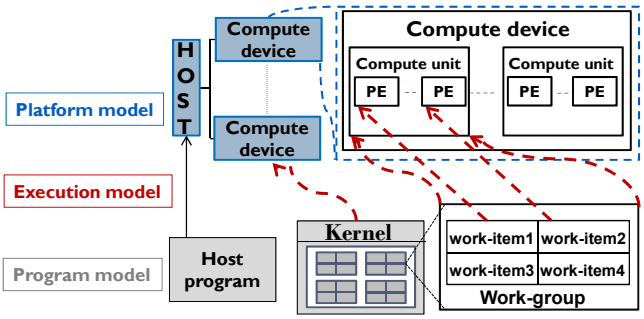


Fig. 1: Overview of the OpenCL architecture.

The work presented in [3] proposes a framework for multicore CPUs that provides a compilation environment for the host program as well as the OpenCL kernels. A performance evaluation is demonstrated by executing two different standalone applications on a specific quad-core processor.

Another evaluation environment is demonstrated in [4] where a complete study is presented to evaluate a number of OpenCL parameters including the API overhead, the OpenCL execution model, vectorization etc. The results are evaluated for general purpose computing by using a number of independent benchmarks on a desktop platform.

Another application-centric evaluation environment is presented in [5] where the applications are evaluated for general purpose computing on multicore CPUs. Interestingly, the results are compared with alternative designs, namely OpenMP-based implementations of the same applications.

Therefore, the existing approaches are limited to the evaluation of OpenCL for general purpose computing, i.e., they use benchmarks which are evaluated on desktop computers. In contrast, our approach considers a systematic evaluation of OpenCL for embedded parallel computing in a distributed automotive embedded system.

III. PROPOSED APPROACH

As discussed, OpenCL distinguishes between a host and kernels where the host is a centralized entity that is responsible for managing the execution of the kernels on the available target devices. The proposed approach adopts this idea in that it enables the user to utilize the available computing resources using a combination of different scheduling and mapping schemes. An overview of the proposed approach is depicted in Fig. 2.

A. User Interface and Runtime Manager

A user interface (UI) is designed to allow the user to intuitively set up and to configure the desired evaluation environment. A user can select OpenCL kernels from an extendable list of available kernels along with their data dependencies. The UI also displays a list of available target devices where the user can select a particular device for execution. Furthermore, a user

Authorized licensed use limited to: University of Glasgow. Downloaded on August 30,2023 at 12:46:10 UTC from IEEE Xplore. Restrictions apply.

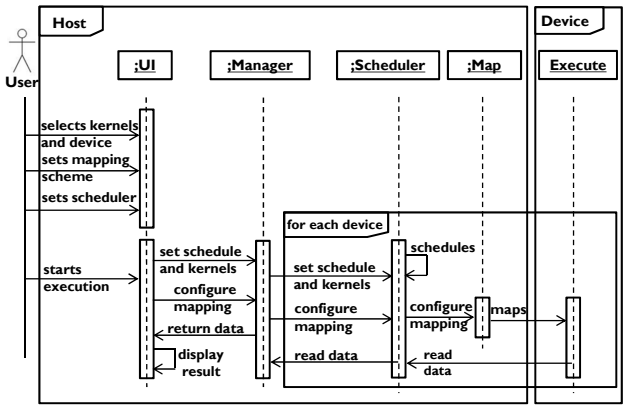


Fig. 2: Overview of the proposed approach.

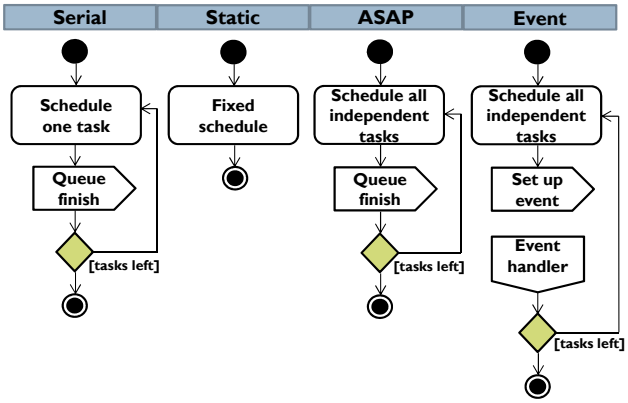


Fig. 3: Available scheduling schemes.

can configure different mapping schemes mainly by selecting a combination of different OpenCL parameters such as number of work-items, work-groups, etc. Finally, the UI provides a list of available scheduling schemes where a particular scheme can be selected that will be used in combination with the already chosen mapping configuration. However, depending on the underlying scheme of the selected scheduler, it may not allow all mapping combinations. Upon invoking a start button, the *Runtime Manager* (RM) starts the execution based on the selected evaluation environment.

Based on that, the RM sets up and initializes the kernels, sets up the OpenCL mapping configuration, and finally invokes the selected scheduler. Once the complete execution of the selected evaluation environment is finished, the RM reads back the evaluated results and supplies it to the UI. Intuitively, the UI is capable of displaying and plotting the results on the fly. In addition, the results are logged in a csv file for offline analysis and monitoring.

B. Scheduling and Mapping

The chosen scheduler schedules the selected kernels (tasks) based on the underlying scheduling scheme. To this end, four distinctive data driven scheduling schemes are developed and made available to the user as shown in Fig. 3. The *serial scheduler*, as the name suggests, realizes a serial execution of tasks, i.e., one task at a time. It implements a dynamic scheduling scheme that enqueues one task at a time for execution and waits until it is completely executed. The *ASAP scheduler* also relies on a dynamic scheduling scheme that exploits task-level parallelism by enqueuing all independent tasks at a time. It then waits until all enqueued tasks are completely executed. The *static scheduler* implements a static scheme that follows a predefined fixed schedule of tasks. The current implementation of the static scheduler (for the case study) is simply a static version of the ASAP scheduler, however, it avoids the runtime overhead of dynamically searching the independent tasks. Finally, the *Event scheduler* also implements a dynamic scheduling scheme, and is similar to the ASAP scheduler, except that it does not wait for the complete execution of enqueued tasks. Instead, the Event scheduler sets up an event for all scheduled (enqueued) tasks, and therefore will be automatically notified when the execution terminates. Hence, it provides a more flexible scheme by allowing itself to perform other operations (if needed) while the already scheduled tasks are being executed.

Regardless of which scheduler is selected by the user, each scheduled task is then mapped on the target device for execution based on the configured mapping scheme. A mapping scheme is configured by selecting a combination of different parameters including the numbers of data samples, work-items, work-groups, etc. This enables the user to potentially exploit the data-level parallelism of the scheduled task by using the desired mapping scheme. A combination of different scheduling and mapping schemes can be used to exploit both the coarse-grained task-level parallelism and fine-grained data-level parallelism, respectively. Hence, the proposed approach systematically facilitates the substantial acceleration in parallel computing.

IV. CASE STUDY

This section first introduces the considered distributed embedded system, and then presents the proposed application setup, prior to presenting the detailed evaluations.

A. The ConceptCar

The *ConceptCar* [6], [7] (designed and developed by our group) is an experimental embedded system with the objective of testing and verifying car features by deploying different classes of applications. The ConceptCar, although not as big as a conventional car, has been built and engineered as close to a modern car as possible.

1) Hardware Design: The ConceptCar is a research platform remotely operated via a standard 2-channel (throttle and steering) 27MHz radio transmitter system. It incorporates a set of sensors (wheel speed, gyro/accelerometer, distance etc.) for interacting with the environment and surroundings. It uses an air-cooled sensorless brushless electrical motor for driving, and a servo motor for steering. The power train of the ConceptCar features two independent power sources: one for the heavy load electric system (motors/actuators), and another one for powering up all the ECUs.

Authorized licensed use limited to: University of Glasgow. Downloaded on August 30,2023 at 12:46:10 UTC from IEEE Xplore. Restrictions apply.

2) Computational Architecture: Although not incorporated with as many ECUs as a modern car can carry, the ConceptCar still features 7 different ECUs, as shown in Fig. 4. These ECUs are organized in three processing units. The *SensorBoard ECUs*, as incorporated with different sensors, form the input processing unit which is responsible for interacting with the environment. The *multicore core ECU* is used as a data processing unit and only comes into play when complex mathematical computations are required. The *ActorBoard ECU* forms the output processing unit and is responsible for creating the PWM signals to drive the actuators (dc motor and servo). Similar to a modern car, all ECUs interact with each other via a centralized CAN bus architecture. Since the ConceptCar incorporates two separate power supplies, a special ECU called EmergencyBoard is added which mainly isolates the actuators from the other boards (galvanic isolation).

In contrast to other ECUs, the multicore ECU features a heterogeneous computing platform consisting of a quadcore CPU and an integrated GPU, and is used for parallel computing. It has therefore been used as the target embedded platform for this particular case study.

B. Application Setup

For this case study, we designed and developed a number of related ADAS applications under a common application setup as shown in Fig. 5. The main focus of this work is not to propose efficient algorithms for ADAS applications. Instead, the idea is to develop a set of related applications under a common system using a minimal set of sensors, mainly for initial lab testing and performance evaluations.

The proposed application setup can be understood in a three-level design where the system first retrieves raw data of sensors from the CAN bus which is forwarded to the second level. Seven different sensors have been used including a temperature sensor, a dual-axis accelerometer, wheel speed sensors, and others. The second level implements data acquisition (DAQ) applications for signal conditioning and digital value conversions of the used sensors. Specifically, four different DAQ applications are implemented to provide the converted numeric values and other optimized results for temperature, acceleration, wheel speed, and battery voltage. The third level, fed by the applications from the second level, implements different ADAS applications. For brevity, we briefly discuss each implemented application as shown in Fig. 5.

The *TempRange* application keeps track of the device temperature and sends a warning signal to the system if the temperature goes above or below the user-defined limits. The *AccidentDetect* application looks for abnormal acceleration spikes to detect accidents and unexpected movements. The measured spikes are examined against the predefined thresholds that are not achievable under normal circumstances. The *CruiseControl* application computes the throttle offset and recommends it to the system, namely the ActorBoard ECU. This offset is used to adjust the throttle value of the ConceptCar and maintain a constant speed provided by the user. The *DistanceTracker* simply

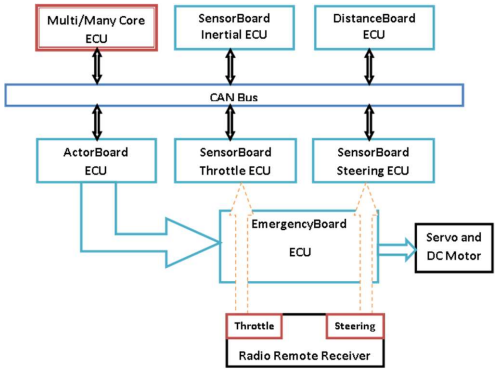


Fig. 4: Architecture of the ConceptCar.

keeps track of the driven distance. In contrast, the *DriveRange* application calculates the maximum drivable distance of the ConcepCar based on the current level of the battery voltage. The *TractionMonitor* application monitors the traction status of the rear wheels and warns the driver if suspects fishtailing, i.e., the rear wheels start losing traction. Finally, the *TurningRadius* application computes and displays the turning radius of the ConceptCar when driving around a corner.

C. Evaluation

To validate the usability of the proposed approach, this case study considers a detailed evaluation of the presented features.

1) Evaluation Environment: The proposed application setup, as shown in Fig. 5, is evaluated on the target platform, i.e., the *multicore ECU* of the ConceptCar. The multicore core ECU features a heterogeneous computing platform with the following hardware specification:

- CPU: 1.2 GHz quad-core ARM Cortex A53
- GPU: 400 MHz Broadcom VideoCore IV
- 1 GB LPDDR2-900 SDRAM

Furthermore, the software environment used for the evaluation is summarized as follows:

- OpenCL CPU implementation: portable computing language (POCL) v1.3
- OpenCL GPU implementation: VideoCore IV OpenCL (VC4CL) conformant to OpenCL v1.2
- Operating system (OS): Debian GNU/Linux 9.8

The application setup is either executed on the CPU or the GPU at a time with different combinations of scheduling and mapping schemes for evaluating end-to-end performance on individual target devices. To this end, the total execution time of the complete application setup to process the desired number of sensors data samples is used as the performance metric. A maximum of 1,044,000 data samples are used and the average of fifty repetitions is taken for each evaluation.

2) Results: In this section, we first begin by demonstrating the effect of varying individual mapping parameters with each scheduler on the performance. We considered three individual mapping parameters, namely the workload, the work-items and

Authorized licensed use limited to: University of Glasgow. Downloaded on August 30,2023 at 12:46:10 UTC from IEEE Xplore. Restrictions apply.

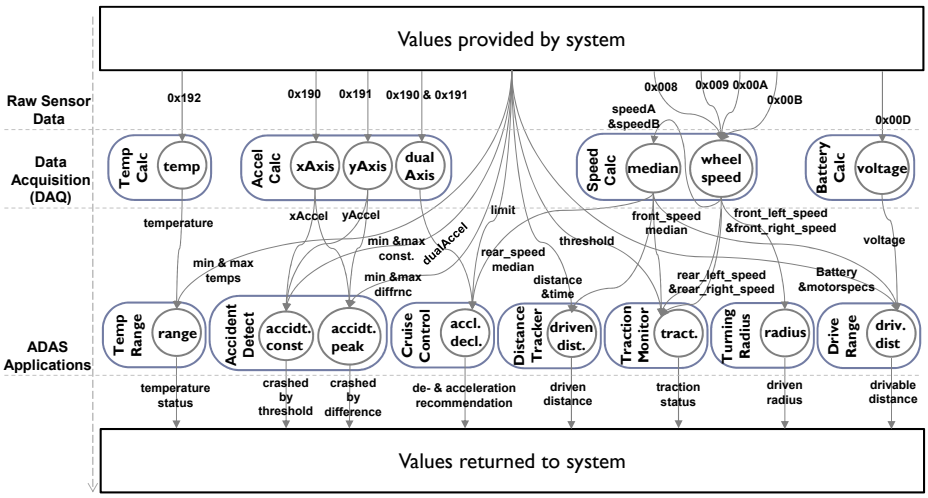


Fig. 5: Proposed application setup.

the work-groups. In the end, we present the best combinations of scheduling and mapping schemes that provided the best end-to-end performance, i.e., the fastest execution times.

a) Workload: The first mapping parameter considered is the workload, i.e, the total number of data samples required to be processed by each mapped execution of the application setup on the device. For this evaluation, the workload is varied while keeping the other mapping parameters fixed. In particular, a single work-item and a single work-group is used, and hence, no data-level parallelism is considered. This evaluation therefore purely compares the performances of the used scheduling schemes, as shown in Fig. 6. Depending on the used implementation, OpenCL allows different workload limits for the target CPU and the GPU, as shown in Fig. 6a and Fig. 6b.

Discussion. Regardless of which scheduling scheme is used, the total execution time increases with the workload as shown in Fig. 6. Overall, the static scheduler performed best of all schedulers. This effect can be clearly observed as the workload is increased from the minimum value of one up to the maximum value of over one million data samples. This is mainly because the underlying scheme of the static scheduler is especially devised for the proposed application setup at compile time. The serial scheduler does not exploit any task-level parallelism, and therefore performed slowest of all schedulers. The ASAP scheduler and the Event scheduler induced the overhead of dynamically scheduling tasks at runtime. Additionally, the Event scheduler further caused the overhead of setting up events and calling handlers at runtime. These overheads therefore resulted in elevating the execution times.

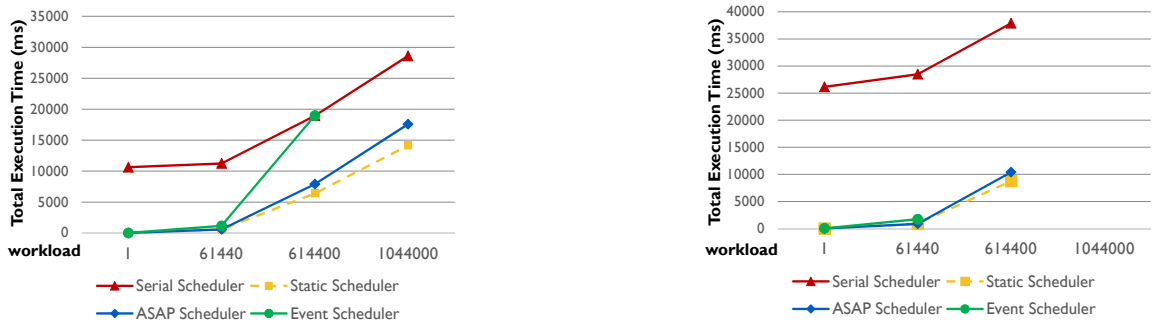
On the CPU, as shown in Fig. 6a, for the maximum workload, the static scheduler is twice as fast as the serial scheduler and the Event scheduler, and about 25% faster than the ASAP scheduler. On the GPU, as shown in Fig. 6b, the same effect can be observed where the static scheduler being the fastest, is up to more than 332% faster than the slowest serial scheduler. The schedulers generally performed faster on the CPU than on the GPU. This is because the communication overhead of OpenCL on GPU is higher than on the CPU. In contrast to the CPU where the host and the kernels reside on the same device, in the case of GPU, the data has to be transferred to the GPU and back to the main memory (host). This overhead therefore contributes in substantially elevating the total execution time. In particular, the static scheduler on the CPU is up to 36% faster than on the GPU for the same workload.

b) Number of work-items: This evaluation analyzes the impact of varying the number of work-items on the performance, using a fixed workload and a fixed single work-group for each device. The idea is to evaluate the effect of increasing the number of parallel instances grouped in a single work-group with individual scheduling schemes as shown in Fig. 7. Hence, this evaluation considers both the task-level parallelism as provided by the scheduling schemes and a limited data-level parallelism confined to a single work-group.

Discussion. The number of work-items affects performance differently on CPUs and GPUs, based on different architectural characteristics as well as on the used OpenCL implementations. In general, since GPUs generally accommodate a large number of streaming cores/CUDA cores, they are capable of managing a large number of parallel threads. On the contrary, the ability of a CPU to handle parallel threads is just limited to the number of few available cores.

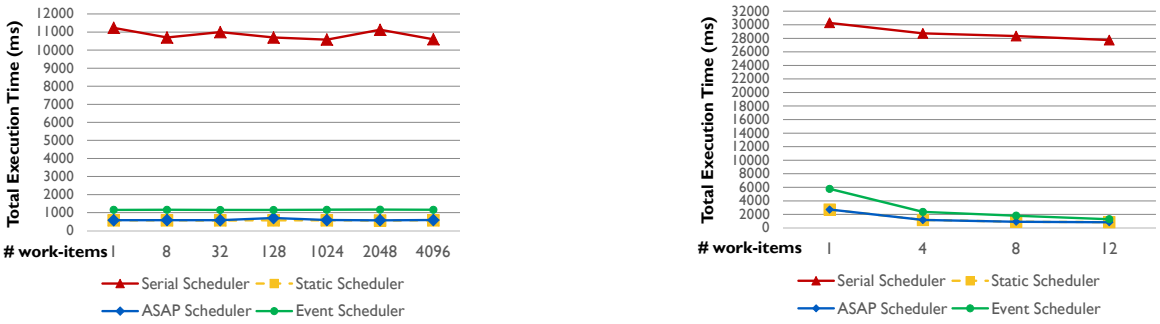
Surprisingly, increasing the number of work-items on the CPU imposed a little or no impact on the total execution time with all schedulers (see Fig. 7a). Based on our profiling analysis, we discovered that the POCL runtime used for CPU automatically coalesces the work-items in a single thread, and hence shows no impact on the performance. On the GPU, as shown in Fig. 7b, increasing the number of work-items with each scheduler substantially improves the performance. In particular, with the highest number of work-items used (i.e., twelve), the Event scheduler improved the performance

Authorized licensed use limited to: University of Glasgow. Downloaded on August 30,2023 at 12:46:10 UTC from IEEE Xplore. Restrictions apply.



(a) CPU (b) GPU

Fig. 6: Workload vs. total execution time.



(a) CPU (b) GPU

Fig. 7: Work-items vs. total execution time.

by more than 347%, the ASAP scheduler improved by almost 223%, the static scheduler improved by more than 216%, and the serial scheduler improved by almost 10%. Overall, the static and the ASAP stand the fastest of all schedulers with almost 33 times faster than the slowest serial scheduler.

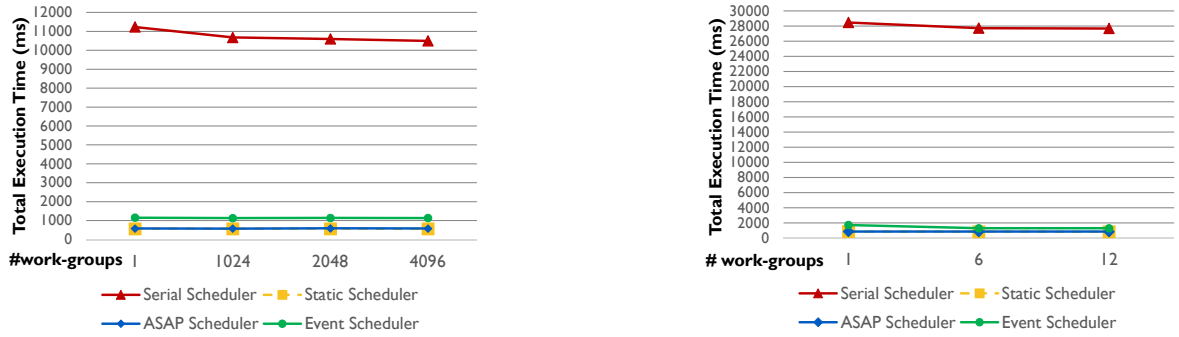
c) Number of work-groups: In general, on GPUs, a work-group or multiple work-groups are executed on a streaming multiprocessor (SM) where ideally each work-item is allocated to a streaming core/SIMD lane. On CPUs, generally, a work-group is allocated to a logical core. The target CPU and the GPU have different work-group size limits. This evaluation analyzes the impact of increasing the number of work-groups with individual scheduling schemes on the performance, as shown in Fig. 8. A fixed workload and a fixed number of work-items are used for each device. The fixed work-items are grouped differently depending on the number of work-groups used. Hence, the entire parallelism is considered, i.e., the task-level parallelism as provided by the scheduling schemes and the data-level parallelism provided by multiple work-items grouped in multiple work-groups.

Discussion. On the CPU, as shown in Fig. 8a, increasing the number of work-groups has little or no impact on the total execution time with all schedulers. This is mainly because the POCL runtime simply sequentializes the execution of work-groups on the CPU, and hence shows no effect on the performance. On the contrary, on the GPU, as shown in Fig. 8b, increasing the number of work-groups with all schedulers further improves the performance from the last evaluation of work-items. To this end, with the highest number of work-groups used (i.e., twelve), the Event scheduler improved the performance by more than 33%, the serial scheduler improved by almost 3%, the static scheduler improved by almost 2%, and the ASAP scheduler improved by almost 1.6%. Overall, the static scheduler proved to be the fastest of all schedulers, with more than 33 times faster than the slowest serial scheduler.

d) Best scheduling and mapping combinations: Based on the general detailed evaluations, we derived the top three combinations of scheduling and mapping schemes on each device that provided the fastest execution times, as shown in Fig. 9. Based on the results, all schedulers achieved their best on the CPU with a single work-item and a single work-group. However, on the GPU, all schedulers achieved their fastest execution times with a total number of twelve work-items, and a total number of twelve work-groups. Thereby suggesting that the exploitation of entire parallelism, i.e., the task-level as well as the data-level parallelism on the GPU resulted in fastest execution times.

On the CPU, the static scheduler proved to be the fastest, in particular, more than twice as fast as the Event scheduler, and about 1.9% faster than the ASAP scheduler. Similarly, on the GPU, the static scheduler and the ASAP scheduler performed comparably fast, with the static one only slightly

Authorized licensed use limited to: University of Glasgow. Downloaded on August 30,2023 at 12:46:10 UTC from IEEE Xplore. Restrictions apply.



(a) CPU (b) GPU

Fig. 8: Work-groups vs. total execution time.

faster than the ASAP. To this end, both the static scheduler and the ASAP scheduler are approximately 53% faster than the Event scheduler. Finally, the static scheduler being the fastest, performed about 47% faster on the CPU than on the GPU. To this end, it is important to observe that based on the used application setup, the task-level parallelism clearly dominated over the data-level parallelism. Therefore, the application setup on the GPU understandably performed slower than on the CPU because the communication overhead of OpenCL on GPU is higher than the performance gain of executing instances in parallel on multicores.

V. CONCLUSIONS AND FUTURE WORK

We presented an approach for employing OpenCL as a standard hardware abstraction that enables the user to systematically utilize the available computing resources using a combination of different scheduling schemes and mapping configurations. In particular, we illustrated the approach by a case study of a distributed automotive embedded system, namely the ConceptCar. Detailed evaluations are presented to demonstrate the features of our approach.

Based on the results, we observed that the static scheduler, designed especially for the used application setup, performed substantially better than the other dynamic scheduling schemes. Second, we observed that the used POCL runtime implementation on the CPU does not fully exploit the data-level parallelism as offered by the OpenCL specification. It is also evaluated that the application setup executed fastest on the GPU, when both the task-level as well the data-level parallelism are exploited. Finally, the application setup generally executed faster on the CPU than on the GPU, mainly because the communication overhead of OpenCL for GPUs is higher than the performance gain of executing instances in parallel on multicores.

Our future work aims at the further exploration of scheduling and mapping schemes for heterogeneous platforms, i.e., using multiple devices together at a time.

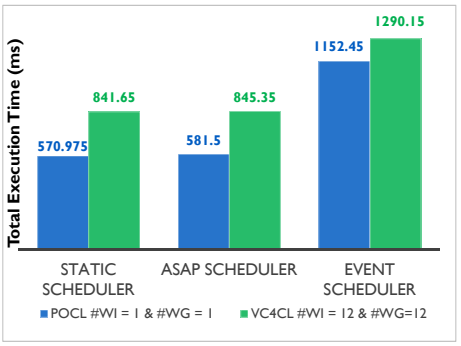


Fig. 9: Top three combinations on each device.

REFERENCES

[1] S. Samii, A. Cervin, P. Eles, and Z. Peng, "Integrated scheduling and synthesis of control applications on distributed embedded systems," in *Design, Automation and Test in Europe*. Nice, France: IEEE, 2009, pp. 57–62.

[2] J. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, May-June 2010.

[3] J.-H. Hong, Y. Ahn, B. Kim, and K.-S. Chung, "Design of opencl framework for embedded multi-core processors," vol. 60, no. 2, May 2014, pp. 233–241.

[4] J. Lee, N. Nigania, H. Kim, K. Patel, and H. Kim, "OpenCL performance evaluation on modern multicore CPUs," *Scientific Programming*, pp. 859 491:1–859 491:20, January 2015.

[5] J. Shen, J. Fang, H. Sips, and A. Varbanescu, "An application-centric evaluation of OpenCL on multi-core CPUs," *Parallel Computing*, vol. 39, no. 12, pp. 834–850, December 2013.

[6] O. Rafique, M. Gesell, and K. Schneider, "Learning various aspects of a distributed real-time automotive embedded system," in *Workshop on Embedded and Cyber-Physical Systems Education (WESE)*, P. Marwedel, J. Jackson, and K. Ricks, Eds. Montreal, Canada: ACM, 2013.

[7] ——, "Generating hardware specific code at different abstraction levels using Averest," in *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, H. Corporaal and S. Stuijk, Eds. Sankt Goar, Germany: ACM, 2013, pp. 90–92.

[8] V. Kukkala, J. Tunnell, S. Pasricha, and T. Bradley, "Advanced driver-assistance systems: A path toward autonomous vehicles," *IEEE Consumer Electronics Magazine*, vol. 7, pp. 18–25, Sep 2018.

Authorized licensed use limited to: University of Glasgow. Downloaded on August 30,2023 at 12:46:10 UTC from IEEE Xplore. Restrictions apply.